

# Automatically Generating Features for Learning Program Analysis Heuristics

Kwonsoo Chae   Hakjoo Oh

Korea University  
{kchae,hakjoo\_oh}@korea.ac.kr

Kihong Heo

Seoul National University  
khheo@ropas.snu.ac.kr

Hongseok Yang

University of Oxford  
hongseok.yang@cs.ox.ac.uk

## Abstract

We present a technique for automatically generating features for data-driven program analyses. Recently data-driven approaches for building a program analysis have been proposed, which mine existing codebases and automatically learn heuristics for finding a cost-effective abstraction for a given analysis task. Such approaches reduce the burden of the analysis designers, but they do not remove it completely; they still leave the highly nontrivial task of designing so called features to the hands of the designers. Our technique automates this feature design process. The idea is to use programs as features after reducing and abstracting them. Our technique goes through selected program-query pairs in codebases, and it reduces and abstracts the program in each pair to a few lines of code, while ensuring that the analysis behaves similarly for the original and the new programs with respect to the query. Each reduced program serves as a boolean feature for program-query pairs. This feature evaluates to true for a given program-query pair when (as a program) it is included in the program part of the pair. We have implemented our approach for three real-world program analyses. Our experimental evaluation shows that these analyses with automatically-generated features perform comparably to those with manually crafted features.

## 1. Introduction

In an ideal world, a static program analysis adapts to a given task automatically, so that it uses expensive techniques for improving analysis precision only when those techniques are absolutely necessary. In a real world, however, most static analyses are not capable of doing such automatic adaptation. Instead, they rely on fixed manually-designed heuristics for deciding when these precision-improving but costly techniques should be applied. These heuristics are usually suboptimal and brittle. More importantly, they are the outcomes of a substantial amount of laborious engineering efforts of analysis designers.

Addressing these concerns with manually-designed heuristics has been the goal of a large body of research in the program-analysis community [4, 11, 17, 19–21, 25, 35, 37, 51, 53, 54]. Recently researchers started to explore data-

driven approaches, where a static analysis uses a parameterized heuristic and the parameter values that maximize the analysis performance are learned automatically from existing codebases via machine learning techniques [10, 18, 22, 40]; the learned heuristic is then used for analyzing previously unseen programs. The approaches have been used to generate various cost-effective analysis heuristics automatically, for example, for controlling the degree of flow or context-sensitivity [40], or determining where to apply relational analysis [22], or deciding the threshold values of widening operators [10].

However, these data-driven approaches have one serious drawback. Their successes crucially depend on the qualities of so called features, which convert analysis inputs, such as programs and queries, to the kind of inputs that machine learning techniques understand. Designing a right set of features requires a nontrivial amount of knowledge and efforts of domain experts. Furthermore, the features designed for one analysis do not usually generalize to others. For example, in [40], a total of 45 features were manually designed for controlling flow-sensitivity, but a new set of 38 features were needed for controlling context-sensitivity. This manual task of crafting features is a major impediment to the widespread adoption of data-driven approaches in practice, as in other applications of machine learning techniques.

In this paper, we present a technique for automatically generating features for data-driven static program analyses. From existing codebases, a static analysis with our technique learns not only an analysis heuristic but also features necessary to learn the heuristic itself. In the first phase of this learning process, a set of features appropriate for a given analysis task is generated from given codebases. The next phase uses the generated features and learns an analysis heuristic from the codebases. Our technique is underpinned by two key ideas. The first idea is to run a generic program reducer (e.g., C-Reduce [46]) on the codebases with a static analysis as a subroutine, and to synthesize automatically *feature programs*, small pieces of code that minimally describe when it is worth increasing the precision of the analysis. Intuitively these feature programs capture programming patterns whose analysis results benefit greatly

from the increased precision of the analysis. The second idea is to generalize these feature programs and represent them by abstract data-flow graphs. Such a graph becomes a boolean predicate on program slices, which holds for a slice when the graph is included in it. We incorporate these ideas into a general framework that is applicable to various parametric static analyses.

We show the effectiveness and generality of our technique by applying it to three static analyses for the C programming language: partially flow-sensitive interval and pointer analyses, and partial Octagon analysis. Our technique successfully generated features relevant to each analysis, which were then used for learning an effective analysis heuristic. The experimental results show that the heuristics learned with automatically-generated features have performance comparable to those with hand-crafted features by analysis experts.

**Contributions** We summarize our contributions below.

- We present a framework for automatically generating features for learning analysis heuristics. The framework is general enough to be used for various kinds of analyses for the C programming language such as interval, pointer, and Octagon analyses.
- We present a novel method that uses a program reducer for generating good feature programs, which capture important behaviors of static analysis.
- We introduce the notion of abstract data-flow graphs and show how they can serve as generic features for data-driven static analyses.
- We provide extensive experimental evaluations with three different kinds of static analyses.

**Outline** We informally describe our approach in Section 2. The formal counterparts of this description take up the next four sections: Section 3 for the definition of parametric static analyses considered in the paper, Section 4 for an algorithm that learns heuristics for choosing appropriate parameter values for a given analysis task, Section 5 for our technique for automatically generating features, and Section 6 for instance analyses designed according to our approach. In Section 7, we report the findings of the experimental evaluation of our approach. In Sections 8 and 9, we explain the relationship between our approach and other prior works, and finish the paper with concluding remarks.

## 2. Overview

We illustrate our approach using its instantiation with a partially flow-sensitive interval analysis.

Our interval analysis is query-based and partially flow-sensitive. It uses a classifier  $\mathcal{C}$  that predicts, for each query in a given program, whether flow-sensitivity is crucial for proving the query: the query can be proved with flow-sensitivity but not without it. If the prediction is positive, the analysis applies flow-sensitivity (FS) to the program variables

that may influence the query: it computes the data-flow slice of the query and tracks the variables in the slice flow-sensitively. On the other hand, if the prediction is negative, the analysis applies flow-insensitivity (FI) to the variables on which the query depends.

For example, consider the following program:

```

1 x = 0; y = 0; z = input(); w = 0;
2 y = x; y++;
3 assert (y > 0); // Query 1
4 assert (z > 0); // Query 2
5 assert (w == 0); // Query 3

```

The first query needs FS to prove, and the second is impossible to prove because the value of  $z$  comes from the external input. The last query is easily proved even with FI. Ideally, we want the classifier to give positive prediction only to the first query, so that our analysis keeps flow-sensitive results only for the variables  $x$  and  $y$ , on which the first query depends, and analyzes other variables flow-insensitively. That is, we want the analysis to compute the following result:

flow-sensitive result		flow-insensitive result
line	abstract state	abstract state
1	$\{x \mapsto [0, 0], y \mapsto [0, 0]\}$	$\{z \mapsto \top, w \mapsto [0, 0]\}$
2	$\{x \mapsto [0, 0], y \mapsto [1, 1]\}$	
3	$\{x \mapsto [0, 0], y \mapsto [1, 1]\}$	
4	$\{x \mapsto [0, 0], y \mapsto [1, 1]\}$	
5	$\{x \mapsto [0, 0], y \mapsto [1, 1]\}$	

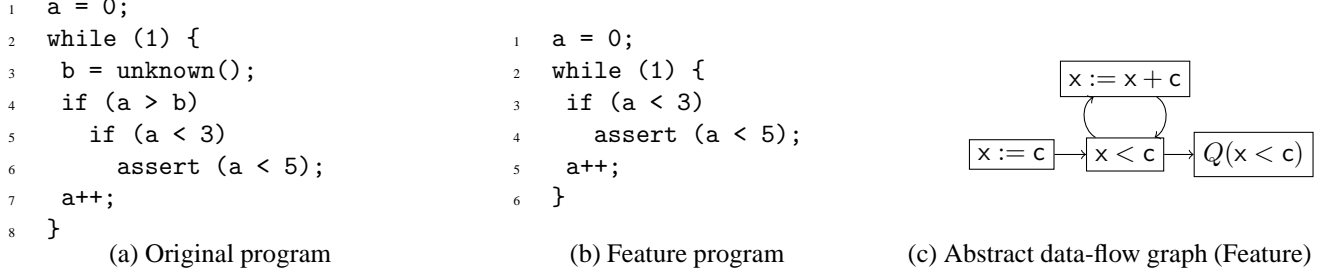
Note that for  $x$  and  $y$ , the result keeps a separate abstract state at each program point, but for the other variables  $z$  and  $w$ , it has just one abstract state for all program points.

### 2.1 Learning a Classifier

The performance of the analysis crucially depends on the quality of its classifier  $\mathcal{C}$ . Instead of designing the classifier manually, we learn it from given codebases automatically. Let us illustrate this learning process with a simple codebase of just one program  $P$ .

The input to the classifier learning is the collection  $\{(q^{(i)}, b^{(i)})\}_{i=1}^n$  of queries in  $P$  labeled with values 0 and 1. The label  $b^{(i)}$  indicates whether the corresponding query  $q^{(i)}$  can be proved with FS but not with FI. These labeled data are automatically generated by analyzing the codebase  $\{P\}$  and identifying the queries that are proved with FS but not with FI.

Given such data  $\{(q^{(i)}, b^{(i)})\}_{i=1}^n$ , we learn a classifier  $\mathcal{C}$  in two steps. First, we represent each query  $q^{(i)}$  by a *feature vector*, which encodes essential properties of the query  $q^{(i)}$  in the program  $P$  and helps learning algorithms to achieve good generalization. Formally, we transform the original data  $\{(q^{(i)}, b^{(i)})\}_{i=1}^n$  to  $\{(v^{(i)}, b^{(i)})\}_{i=1}^n$ , where  $v^{(i)} \in \mathbb{B}^k = \{0, 1\}^k$  is a binary feature vector of query  $q^{(i)}$ . The dimension  $k$  of feature vectors denotes the number of features. Second, to this transformed data set  $\{(v^{(i)}, b^{(i)})\}_{i=1}^n$ , we ap-



**Figure 1.** Example program and feature.

ply an off-the-shelf classification algorithm (such as decision tree) and learn a classifier  $\mathcal{C} : \mathbb{B}^k \rightarrow \mathbb{B}$ , which takes a feature vector of a query and makes a prediction.

The success of this learning process relies heavily on how we convert queries to feature vectors. If the feature vector of a query ignores important information about the query for prediction, learning a good classifier is impossible irrespective of learning algorithms used. In previous work [10, 22, 40], this feature engineering is done manually by analysis designers. For a specific static analysis, they defined a set of *features* and used them to convert a query to a feature vector. But as in other applications of machine learning, this feature engineering requires considerable domain expertise and engineering efforts. Our goal is to automatically generate high-quality features for this program-analysis application.

## 2.2 Automatic Feature Generation

We convert queries to feature vectors using a set of features  $\Pi = \{\pi_1, \dots, \pi_k\}$  and a procedure match. A feature  $\pi_i$  encodes a property about queries. The match procedure takes a feature  $\pi$ , a query  $q_0$  and a program  $P_0$  containing the query, and checks whether the slice of  $P_0$  that may affect  $q_0$  satisfies the property encoded by  $\pi$ . If so, it returns 1, and otherwise, 0. Using  $\Pi$  and match, we transform every query  $q$  in the program  $P$  of our codebase into a feature vector  $v$ :

$$v = \langle \text{match}(\pi_1, q, P), \dots, \text{match}(\pi_k, q, P) \rangle.$$

### 2.2.1 Feature Generation

The unique aspect of our approach lies in our technique for generating the feature set  $\Pi$  from given codebases automatically.<sup>1</sup> Two ideas make this automatic generation possible.

**Generating Feature Programs Using a Reducer** The first idea is to use a generic program reducer. A reducer (e.g., C-Reduce [46]) takes a program and a predicate, and iteratively removes parts of the program as long as the predicate holds.

We use a reducer to generate a collection of small code snippets that describe cases where the analysis can prove a query with FS but not with FI. We first collect a set of queries

in codebases that require FS to prove. Then, for every query in the set, we run the reducer on the program containing the query under the predicate that the query in the reduced program continues to be FS-provable but FI-unprovable. The reducer removes all the parts from the program that are irrelevant to the FS-provability and the FI-unprovability of the query, leading to a *feature program*.

For example, consider the example program in Figure 1(a). The assertion at line 6 can be proved by the flow-sensitive interval analysis but not by the flow-insensitive one; with FS, the value of  $a$  is restricted to the interval  $[0, 3]$  because of the condition at line 5. With FI,  $a$  has  $[0, +\infty]$  at all program points. We reduce this program as long as the flow-sensitive analysis proves the assertion while the flow-insensitive one does not, resulting in the program in Figure 1(b). Note that the reduced program only contains the key reasons (i.e., loop and `if (a < 3)`) for why FS works. For example, the command `if (a > b)` is removed because even without it, the flow-sensitive analysis proves the query. Running the reducer this way automatically removes these irrelevant parts of the original program.

In experiments, we used C-Reduce [46], which has been used for generating small test cases that trigger compiler bugs. The original program in Figure 1(a) is too simplistic and does not fully reflect the amount of slicing done by C-Reduce for real programs. In our experiments, we found that C-Reduce is able to transform programs with >1KLOC to those with just 5–10 lines, similar to the one in Figure 1(b).

### Representing Feature Programs by Abstract Data-flow Graphs

The second idea is to represent the feature programs by abstract data-flow graphs. We build graphs that describe the data flows of the feature programs. Then, we abstract individual atomic commands in the graphs, for instance, by replacing some constants and variables with the same fixed symbols  $c$  and  $x$ , respectively. The built graphs form the collection of features  $\Pi$ .

For example, the feature program in Figure 1(b) is represented by the graph in Figure 1(c). The graph captures the data flows of the feature program that influence the query. At the same time, the graph generalizes the program by abstracting its atomic commands. All the variables are replaced by the same symbol  $x$ , and all integers by  $c$ , which in particu-

<sup>1</sup>In our implementation, we partition the codebases to two groups. Programs in the first group are used for feature generation and learning and those in the other group are used for cross validation.

lar makes the conditions  $a < 3$  and  $a < 5$  the same abstract condition  $x < c$ .

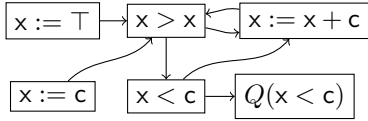
How much should we abstract commands of the feature program? The answer depends on a static analysis. If we abstract commands aggressively, this would introduce a strong inductive bias, so that the algorithm for learning a classifier might have hard time for finding a good classifier for given codebases but would require fewer data for generalization. Otherwise, the opposite situation would occur. Our technique considers multiple abstraction levels, and automatically picks one to a given static analysis using the combination of searching and cross-validation (Section 5).

### 2.2.2 Matching Algorithm

By using the technique explained so far, we generate an abstract data-flow graph for each FS-provable but FI-unprovable query in given codebases. These graphs form the set of features,  $\Pi = \{\pi_1, \dots, \pi_k\}$ .

The match procedure takes a feature (i.e., abstract data-flow graph)  $\pi_i \in \Pi$ , a query  $q_0$ , and a program  $P_0$  containing  $q_0$ . Then, it checks whether the slice of  $P_0$  that may affect  $q_0$  includes a piece of code described by  $\pi_i$ . Consider the query in the original program in Figure 1(a) and the feature  $\pi$  in Figure 1(c). Checking whether the slice for the query includes the feature is done in the following two steps.

We first represent the query in Figure 1(a) itself by an abstract data-flow graph:



Note that the graph is similar to the one in Figure 1(c) but it contains all the parts of the original program. For instance, it has the node  $x > x$  and the edge from this node to  $x < c$ , both of which are absent in the feature. The unknown value, such as the return value of `unknown()`, is represented by  $T$ .

Next, we use a variant of graph inclusion to decide whether the query includes the feature. We check whether every vertex of the feature is included in the graph of the query and whether every arc of the feature is included in the *transitive closure* of the graph. The answers to both questions are yes. For instance, the path for the arc  $x := x + c \rightarrow x < c$  in the feature is  $x := x + c \rightarrow x > x \rightarrow x < c$  in the graph of the query.

Note that we use a variant of graph inclusion where an arc of one graph is allowed to be realized by a *path* of its including graph, not necessarily by an *arc* as in the usual definition. This variation is essential for our purpose. When we check a feature against a query, the feature is reduced but the query is not. Thus, even when the query here is the one from which the feature is generated, this checking is likely to fail if we use the usual notion of graph inclusion (i.e.,  $G_1 = (V_1, E_1)$  is included in  $G_2 = (V_2, E_2)$  iff  $V_1 \subseteq V_2$  and  $E_1 \subseteq E_2$ ). In theory, we could invoke a reducer on the

query, but this is not a viable option because reducing is just too expensive to perform every time we analyze a program. Instead, we take a (less expensive) alternative based on the transitive closure of the graph of the query.

## 3. Setting

**Parametric Static Analysis** We use the setting for parametric static analyses in [29]. Let  $P \in \mathbb{P}$  be a program to analyze. We assume that a set  $\mathbb{Q}_P$  of queries (i.e., assertions) in  $P$  is given together with the program. The goal of the analysis is to prove as many queries as possible. A static analysis is parameterized by a set of program components. We assume a set  $\mathbb{J}_P$  of program components that represent parts of  $P$ . For instance, in our partially flow-sensitive analysis,  $\mathbb{J}_P$  is the set of program variables. The parameter space is defined by  $(\mathcal{A}_P, \sqsubseteq)$  where  $\mathcal{A}_P$  is the binary vector  $\mathbf{a} \in \mathcal{A}_P = \mathbb{B}^{\mathbb{J}_P} = \{0, 1\}^{\mathbb{J}_P}$  with the pointwise ordering. We sometimes regard a parameter  $\mathbf{a} \in \mathcal{A}_P$  as a function from  $\mathbb{J}_P$  to  $\mathbb{B}$ , or the set  $\mathbf{a} = \{j \in \mathbb{J}_P \mid \mathbf{a}_j = 1\}$ . In the latter case, we write  $|\mathbf{a}|$  for the size of the set. We define two constants in  $\mathcal{A}_P$ :  $\mathbf{0} = \lambda j \in \mathbb{J}_P. 0$  and  $\mathbf{1} = \lambda j \in \mathbb{J}_P. 1$ , which represent the most imprecise and precise abstractions, respectively. We omit the subscript  $P$  when there is no confusion. A parametric static analysis is a function  $F : \mathbb{P} \times \mathcal{A} \rightarrow \wp(\mathbb{Q})$ , which takes a program to analyze and a parameter, and returns a set of queries proved by the analysis under the given parameter. In static analysis of C programs, using a more refined parameter typically improves the precision of the analysis but increases the cost.

**Analysis Heuristic that Selects a Parameter** The parameter of the analysis is selected by an analysis heuristic  $\mathcal{H} : \mathbb{P} \rightarrow \mathcal{A}$ . Given a program  $P$ , the analysis first applies the heuristic to  $P$ , and then uses the resulting parameter  $\mathcal{H}(P)$  to analyze the program. That is, it computes  $F(P, \mathcal{H}(P))$ . If the heuristic is good, running the analysis with  $\mathcal{H}(P)$  would give results close to those of the most precise abstraction ( $F(P, \mathbf{1})$ ), while the analysis cost is close to that of the least precise abstraction ( $F(P, \mathbf{0})$ ). Previously, such a heuristic was designed manually (e.g., [25, 37, 54]), which requires a large amount of engineering efforts of analysis designers.

## 4. Learning an Analysis Heuristic

In a data-driven approach, an analysis heuristic  $\mathcal{H}$  is automatically learned from given codebases. In this section, we describe our realization of this approach while assuming that a set of features is given; this assumption will be discharged in Section 5. We denote our method for learning a heuristic by  $\text{learn}(F, \Pi, \mathbf{P})$ , which takes a static analysis  $F$ , a set  $\Pi$  of features, and codebases  $\mathbf{P}$ , and returns a heuristic  $\mathcal{H}$ .

**Learning a Classifier** In our method, learning a heuristic  $\mathcal{H}$  boils down to learning a classifier  $\mathcal{C}$ , which predicts for each query whether the query can be proved by a static analysis with increased precision but not without it. Suppose

that we are given codebases  $\mathbf{P} = \{P_1, \dots, P_n\}$ , a set of features  $\Pi = \{\pi_1, \dots, \pi_k\}$ , and a procedure match. The precise definitions of  $\Pi$  and match will be given in the next section. For now, it is sufficient just to know that a feature  $\pi_i \in \Pi$  describes a property about queries and match checks whether a query satisfies this property.

Using  $\Pi$  and match, we represent a query  $q \in \mathbb{Q}_P$  in a program  $P$  by a feature vector  $\Pi(q, P) \in \mathbb{B}^k$  such that the  $i$ th component of the vector is the result of  $\text{match}(\pi_i, q, P)$ . This vector representation enables us to employ the standard tools for learning and using a binary classifier. In our case, a classifier is just a map  $\mathcal{C} : \mathbb{B}^k \rightarrow \mathbb{B}$  and predicts whether the query can be proved by the analysis under high precision (such as flow-sensitivity) but not with low precision (such as flow-insensitivity). To use such a classifier, we just need to call it with  $\Pi(q, P)$ . To learn it from codebases, we follow the two steps described below:

1. We generate labeled data  $D \subseteq \mathbb{B}^k \times \mathbb{B}$  from the codebases:  $D = \{(\Pi(q, P_i), b_i) \mid P_i \in \mathbf{P} \wedge q \in \mathbb{Q}_{P_i}\}$ , where  $b_i = (q \in F(P_i, 1) \setminus F(P_i, 0))$ . That is, for each program  $P_i \in \mathbf{P}$  and a query  $q$  in  $P_i$ , we represent the query by a feature vector and label it with 1 if  $q$  can be proved by the analysis under the most precise setting but not under the least precise setting. When it is infeasible to run the most precise analysis (e.g., the Octagon analysis), we instead run an approximate version of it. In experiments with the partial Octagon analysis, we used the impact pre-analysis [37] as an approximation.
2. Then, we learn a classifier from the labeled data  $D$  by invoking an off-the-shelf learning algorithm, such as logistic regression, decision tree, and support vector machine.

**Building an Analysis Heuristic** We construct an analysis heuristic  $\mathcal{H} : \mathbb{P} \rightarrow \mathcal{A}$  from a learned classifier  $\mathcal{C}$  as follows:

$$\mathcal{H}(P) = \bigcup \{\text{req}(q) \mid q \in \mathbb{Q}_P \wedge \mathcal{C}(\Pi(q, P)) = 1\}$$

The heuristic iterates over every query  $q \in \mathbb{Q}_P$  in the program  $P$ , and selects the ones that get mapped to 1 by the classifier  $\mathcal{C}$ . For each of these selected queries, the heuristic collects the parts of  $P$  that may affect the analysis result of the query. This collection is done by the function  $\text{req} : \mathbb{Q} \rightarrow \mathcal{A}$ , which satisfies that  $q \in F(P, 1) \implies q \in F(P, \text{req}(q))$  for all queries  $q$  in  $P$ . This function should be specified by an analysis designer, but according to our experience, this is rather a straightforward task. For instance, our instance analyses (namely, two partially flow-sensitive analyses and partial Octagon analysis) implement  $\text{req}$  via a simple dependency analysis. For instance, in our partially flow-sensitive analysis,  $\text{req}(q)$  is just the set of all program variables in the dependency slice of  $P$  for the query  $q$ . The result of  $\mathcal{H}(P)$  is the union of all the selected parts of  $P$ .

## 5. Automatic Feature Generation

We now present the main contribution of this paper, our feature generation algorithm. The algorithm first generates so

called feature programs from given codebases (Section 5.1), and then converts all the generated programs to abstract data-flow graphs (Section 5.2). The obtained graphs enable the match procedure to transform queries to feature vectors so that a classifier can be applied to these queries (Section 5.3). In this section, we will explain all these aspects of our feature generation algorithm.

### 5.1 Generation of Feature Programs

Given a static analysis  $F$  and codebases  $\mathbf{P}$ ,  $\text{gen\_fp}(\mathbf{P}, F)$  generates feature programs in two steps.

First, it collects the set of queries in  $\mathbf{P}$  that can be proved by the analysis  $F$  under high precision (i.e.,  $F(-, 1)$ ) but not with low precision (i.e.,  $F(-, 0)$ ). We call such queries *positive* and the other non-positive queries *negative*. The negative queries are either too hard in the sense that they cannot be proved even with high precision, or too easy in the sense that they can be proved even with low precision. Let  $\mathbf{P}_{\text{pos}}$  be the set of positive queries and their host programs:

$$\mathbf{P}_{\text{pos}} = \{(P, q) \mid P \in \mathbf{P} \wedge q \in \mathbb{Q}_P \wedge \phi_q(P)\}$$

where  $\mathbb{Q}_P$  is the set of queries in  $P$  and  $\phi_q$  is defined by:

$$\phi_q(P) = (q \notin F(P, 0) \wedge q \in F(P, 1)). \quad (1)$$

Second,  $\text{gen\_fp}(\mathbf{P}, F)$  shrinks the positive queries collected in the first step by using a program reducer. A program reducer (e.g., C-Reduce [46]) is a function of the type:  $\text{reduce} : \mathbb{P} \times (\mathbb{P} \rightarrow \mathbb{B}) \rightarrow \mathbb{P}$ . It takes a program  $P$  and a predicate  $\text{pred}$ , and removes parts of  $P$  as much as possible while preserving the original result of the predicate. At the end, it returns a minimal program  $P'$  such that  $\text{pred}(P') = \text{pred}(P)$ . Our procedure  $\text{gen\_fp}(\mathbf{P}, F)$  runs a reducer and shrinks programs in  $\mathbf{P}_{\text{pos}}$  as follows:

$$\mathbf{P}_{\text{feat}} = \{(\text{reduce}(P, \phi_q), q) \mid (P, q) \in \mathbf{P}_{\text{pos}}\}.$$

$\mathbf{P}_{\text{feat}}$  is the collection of the reduced programs paired with queries. We call these programs *feature programs*. Because of the reducer, each feature program contains only those parts related to the reason that high precision is effective for proving its query. Intuitively, the reducer removes noise in the positive examples  $(P, q) \in \mathbf{P}_{\text{pos}}$ , until the examples contain only the reasons that high precision of the analysis helps prove their queries. The result of  $\text{gen\_fp}(\mathbf{P}, F)$  is  $\mathbf{P}_{\text{feat}}$ .

**Improvement 1: Preserving Analysis Results** A program reducer such as C-Reduce [46] is powerful and is able to reduce C programs of thousands LOC to just a few lines of feature programs. However, some additional care is needed in order to prevent C-Reduce from removing too aggressively and producing trivial programs.

For example, suppose we analyze the following code snippet (excerpted and simplified from bc-1.06) with a partially flow-sensitive interval analysis:

```
1 yychar = 1; yychar = input(); //external input
2 if (yychar < 0) exit(1);
```

```

3 if (yychar <= 289)
4   assert(0 <= yychar < 290); // query q
5   yychar++;

```

The predicate  $\phi_q$  in (1) holds for this program. The analysis can prove the assertion at line 4 with flow-sensitivity, because in that case, it computes the interval  $[0, 289]$  for `yychar` at line 4. But it cannot prove the assertion with flow-insensitivity, because it computes the interval  $[-\infty, +\infty]$  for `yychar` that holds over the entire program.

Reducing the program under the predicate  $\phi_q$  may produce the following program:

```
yychar=1; assert(0<=yychar<290); yychar++;
```

It is proved by flow-sensitivity but not by flow-insensitivity. An ordinary flow-insensitive interval analysis computes the interval  $[1, +\infty]$  because of the increment of `yychar` at the end. Thus the resulting program still satisfies  $\phi_q$ . However, this reduced program does not contain the genuine reason that the original program needed flow-sensitivity: in the original program, the `if` commands at lines 2 and 3 are analyzed accurately only under flow-sensitivity, and the accurate analysis of these commands is crucial for proving the assertion.

To mitigate the problem, we run the reducer with a stronger predicate that additionally requires the preservation of analysis result. In the flow-sensitive analysis of our original program, the variable `yychar` has the interval value  $[0, 289]$  at the assertion. In the above reduced program, on the other hand, it has the value  $[1, 1]$ . The strengthened predicate  $\phi'_q$  in this example is:

$$\phi'_q(P) = (\phi_q(P) \wedge \text{value of yychar at } q \text{ is } [0, 289]). \quad (2)$$

Running the reducer with this new predicate results in:

```

1 yychar = input();
2 if (yychar < 0) exit(1);
3 if (yychar <= 289) assert(0 <= yychar < 290);

```

The irrelevant commands (`yychar = 1`, `yychar++`) in the original program are removed by the reducer, but the important `if` commands at lines 2 and 3 remain in the reduced program. Without these `if` commands, it is impossible to satisfy the new condition (2), so that the reducer has to preserve them in the final outcome. This idea of preserving the analysis result during reduction was essential to generate diverse feature programs. Also, it can be applied to any program analysis easily.

### Improvement 2: Approximating Variable Initialization

Another way for guiding a reducer is to replace commands in a program by their overapproximations and to call the reducer on the approximated program. The rationale is that approximating commands would prevent the reducer from accidentally identifying a reason that is too specific to a given query and does not generalize well. Approximation would help remove such reasons, so that the reducer is more likely to find general reasons.

Consider the following code snippet (from `spell-1.0`):

```

1 pos = 0;
2 while (1) { if (!pos) assert(pos==0); pos++; }

```

The flow-sensitive interval analysis proves the assertion because it infers the interval  $[0, 0]$  for `pos`. Note that this analysis result crucially relies on the condition `!pos`. Before the condition, the value of `pos` is  $[0, +\infty]$ , but the condition refines the value to  $[0, 0]$ . However, reducing the program under  $\phi_q$  in (1) leads to the following program:

```
pos=0; assert(pos==0); pos++;
```

This reduced program no longer says the importance of refining an abstract value with the condition `!pos`. Demanding the preservation of the analysis result does not help, because the value of `pos` is also  $[0, 0]$  in the reduced program.

We fight against this undesired behavior of the reducer by approximating commands of a program  $P$  before passing it to the reducer. Specifically, for every positive query  $(P, q)$  and for each command in  $P$  that initializes a variable with a constant value, we replace the constant by `Top` (an expression that denotes the largest abstract value  $\top$ ) as long as this replacement does not make the query negative. For instance, we transform our example to the following program:

```
pos = Top; // 0 is replaced by Top
while (1) { if (!pos) assert(pos==0); pos++; }
```

Note that `pos = 0` is replaced by `pos = Top`. Then, we apply the reducer to this transformed program, and obtain:

```
pos = Top; if (!pos) assert(pos==0);
```

Note that the reduced program keeps the condition `!pos`; because of the change in the initialization of `pos`, the analysis cannot prove the assertion without using the condition `!pos` in the original program.

## 5.2 Transformation to Abstract Data-Flow Graphs

Our next procedure is `build_dfg( $\mathbf{P}_{feat}, \hat{R}$ )`, which converts feature programs in  $\mathbf{P}_{feat}$  to their data-flow graphs where nodes are labeled with the abstraction of atomic commands in those programs. We call such graphs *abstract data-flow graphs*. These graphs act as what people call features in the applications of machine learning. The additional parameter  $\hat{R}$  to the procedure controls the degree of abstraction of the atomic commands in these graphs. A method for finding an appropriate parameter  $\hat{R}$  will be presented in Section 5.4.

**Step 1: Building Data-Flow Graphs** The first step of `build_dfg( $\mathbf{P}_{feat}, \hat{R}$ )` is to build data-flow graphs for feature programs in  $\mathbf{P}_{feat}$  and to slice these graphs with respect to queries in those programs.

The `build_dfg` procedure constructs and slices such data-flow graphs using standard recipes. Assume a feature program  $P \in \mathbf{P}_{feat}$  represented by a control-flow graph  $(\mathbb{C}, \hookrightarrow)$ , where  $\mathbb{C}$  is the set of program points annotated with atomic commands and  $(\hookrightarrow) \subseteq \mathbb{C} \times \mathbb{C}$  the control-flow relation between those program points. The data-flow graph

$R_1: c \rightarrow lv := e \mid lv := alloc(e) \mid assume(e_1 \prec e_2)$   
 $R_2: e \rightarrow c \mid e_1 \oplus e_2 \mid lv \mid \&lv, \quad lv \rightarrow x \mid *e \mid e_1[e_2]$   
 $R_3: \oplus \rightarrow + \mid - \mid * \mid / \mid < \mid < \mid > \mid >, \quad \prec \rightarrow < \mid \leq \mid > \mid \geq \mid \neq$   
 $R_4: c \rightarrow 0 \mid 1 \mid 2 \mid \dots, \quad x \rightarrow x \mid y \mid z \mid \dots$

**Figure 2.** The set  $R$  of grammar rules for C-like languages

for  $P$  reuses the node set  $\mathbb{C}$  of the control-flow graph, but it uses a new arc relation  $\rightsquigarrow$ :  $c \rightsquigarrow c'$  iff there is a def-use chain in  $P$  from  $c$  to  $c'$  on a memory location or variable  $l$  (that is,  $c \hookrightarrow^+ c'$ ,  $l$  is defined at  $c$ ,  $l$  is used at  $c'$ , and  $l$  is not re-defined in the intermediate program points between  $c$  and  $c'$ ). For each query  $q$  in the program  $P$ , its slice  $(\mathbb{C}_q, \rightsquigarrow_q)$  is just the restriction of the data-flow graph  $(\mathbb{C}_q, \rightsquigarrow_q)$  with respect to the nodes that may reach the query (i.e.,  $\mathbb{C}_q = \{c \in \mathbb{C} \mid c \rightsquigarrow^* c_q\}$ ).

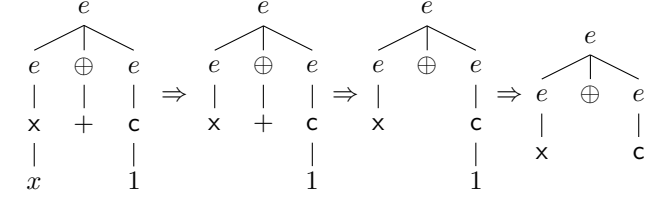
**Step 2: Abstracting Atomic Commands** The second step of  $\text{build\_dfg}(\mathbf{P}_{feat}, \hat{R})$  is to abstract atomic commands in the data-flow graphs obtained in the first step and to collapse nodes labeled with the same abstract command. This abstraction is directed by the parameter  $\hat{R}$ , and forms the most interesting part of the  $\text{build\_dfg}$  procedure.

Our abstraction works on the grammar for the atomic commands shown in Figure 2. The command  $lv := e$  assigns the value of  $e$  into the location of  $lv$ , and  $lv := alloc(e)$  allocates an array of size  $e$ . The assume command  $assume(e_1 \prec e_2)$  allows the program to continue only when the condition evaluates to true. An expression may be a constant integer ( $c$ ), a binary operator ( $e_1 \oplus e_2$ ), an l-value expression ( $lv$ ), or an address-of expression ( $\&lv$ ). An l-value may be a variable ( $x$ ), a pointer dereference ( $*e$ ), or an array access ( $e_1[e_2]$ ).

Let  $R$  be the set of grammar rules in Figure 2. The second parameter  $\hat{R}$  of  $\text{build\_dfg}(\mathbf{P}_{feat}, \hat{R})$  is a subset of  $R$ . It specifies how each atomic command should be abstracted. Intuitively, each rule in  $\hat{R}$  says that if a part of an atomic command matches the RHS of the rule, it should be represented abstractly by the nonterminal symbol in the LHS of the rule. For example, when  $\hat{R} = \{\oplus \rightarrow + \mid -\}$ , both  $x = y + 1$  and  $x = y - 1$  are represented abstractly by the same  $x = y \oplus 1$ , where  $+$  and  $-$  are replaced by  $\oplus$ . Formally,  $\text{build\_dfg}(\mathbf{P}_{feat}, \hat{R})$  transforms the parse tree of each atomic command in  $\mathbf{P}_{feat}$  by repeatedly applying the grammar rules in  $\hat{R}$  backwards to the tree until a fixed point is reached. This transformation loses information about the original atomic command, such as the name of a binary operator. We denote it by a function  $\alpha_{\hat{R}}$ . The following example illustrates this transformation using a simplified version of our grammar.

**Example 1.** Consider the grammar:  $R = \{e \rightarrow x \mid c \mid e_1 \oplus e_2, \quad x \rightarrow x \mid y, \quad c \rightarrow 1 \mid 2, \quad \oplus \rightarrow + \mid -\}$ . Let  $\hat{R} = \{x \rightarrow x \mid y, \quad c \rightarrow 1 \mid 2, \quad \oplus \rightarrow + \mid -\}$ . Intuitively,  $\hat{R}$  specifies that we should abstract variables, constants, and operators in atomic commands and expressions by nonterminals  $x$ ,  $c$ , and  $\oplus$ , respectively. The abstraction is done by applying

rules  $\hat{R}$  backwards to parse trees until none of the rules becomes applicable. For example, the expression  $x + 1$  is abstracted into  $x \oplus c$  as follows:



We first apply the rule  $x \rightarrow x$  backwards to the parse tree (leftmost) and collapse the leaf node  $x$  with its parent. Next, we apply  $\oplus \rightarrow +$  where  $+$  is collapsed to  $\oplus$ . Finally, we apply the rule  $c \rightarrow 1$ , getting the rightmost tree. The result is read off from the last tree, and is the abstract expression  $x \oplus c$ . Similarly,  $y - 2$  gets abstracted to  $x \oplus c$ .  $\square$

For each data-flow slice computed in the first step, our  $\text{build\_dfg}(\mathbf{P}_{feat}, \hat{R})$  procedure applies the  $\alpha_{\hat{R}}$  function to the atomic commands in the slice. Then, it merges nodes in the slice to a single node if they have the same label (i.e., the same abstract atomic command). The nodes after merging inherit the arcs from the original slice. We call the resulting graphs *abstract data-flow graphs*. These graphs describe (syntactic and semantic) program properties, such as the ones used in [40]. For example, the abstract data-flow graph  $(x < c) \rightsquigarrow (x := alloc(x))$  says that a program variable is compared with a constant expression before being used as an argument of a memory allocator (which corresponds to the features #9 and #11 for selective flow-sensitivity in [40]).

We write  $\{\pi_1, \dots, \pi_k\}$  for the abstract data-flow graphs generated by  $\text{build\_dfg}(\mathbf{P}_{feat}, \hat{R})$ . We sometimes call  $\pi_i$  *feature*, especially when we want to emphasize its role in our application of machine learning techniques.

We point out that the performance of a data-driven analysis in our approach depends on the choice of the parameter  $\hat{R}$  to the  $\text{build\_dfg}(\mathbf{P}_{feat}, \hat{R})$  procedure. For example, the Octagon analysis can track certain binary operators such as addition precisely but not other binary operators such as multiplication and shift. Thus, in this case, we need to use  $\hat{R}$  that at least differentiates these two kinds of operators. In Section 5.4, we describe a method for automatically choosing  $\hat{R}$  from data via iterative cross validation.

### 5.3 Abstract Data-flow Graphs and Queries

Abstract data-flow graphs encode properties about queries. These properties are checked by our  $\text{match}_{\hat{R}}$  procedure parameterized by  $\hat{R}$ . The procedure takes an abstract data-flow graph  $\pi$ , a query  $q$  and a program  $P$  that contains the query. Given such inputs, it works in four steps. First,  $\text{match}_{\hat{R}}(\pi, q, P)$  normalizes  $P$  syntactically so that some syntactically different yet semantically same programs become identical. Specifically, the procedure eliminates temporary variables (e.g., convert  $\text{tmp} = b + 1; a = \text{tmp};$  to  $a = b + 1$ ), removes double negations (e.g., convert  $\text{assume}(!!(x==1))$  to  $\text{assume}(x==1)$ ), and makes

---

**Algorithm 1** Automatic Feature Generation

---

**Input:** codebases  $\mathbf{P}$ , static analysis  $F$ , grammar rules  $R$ **Output:** a set of features  $\Pi$ 

```
1: partition  $\mathbf{P}$  into  $\mathbf{P}_{tr}$  and  $\mathbf{P}_{va}$   $\triangleright$  training/validation sets
2:  $\mathbf{P}_{feat} \leftarrow \text{gen\_fp}(\mathbf{P}_{tr}, F)$   $\triangleright$  generate feature programs
3:  $s_{best}, \Pi_{best} \leftarrow -1, \emptyset$ 
4: repeat
5:    $\hat{R} \leftarrow$  choose a subset of  $R$  (i.e.,  $\hat{R} \subseteq R$ )
6:    $\Pi \leftarrow \text{build\_dfg}(\mathbf{P}_{feat}, \hat{R})$   $\triangleright$  Build data-flow graphs
7:    $\mathcal{H}_C \leftarrow \text{learn}(F, \Pi, \mathbf{P}_{tr})$ 
8:    $s \leftarrow \text{evaluate}(F, \mathcal{C}, \mathbf{P}_{va})$   $\triangleright$  Evaluate  $F_1$ -score of  $\mathcal{C}$ 
9:   if  $s > s_{best}$  then
10:      $s_{best}, \Pi_{best} \leftarrow s, \Pi$ 
11:   end if
12: until timeout
13: return  $\Pi_{best}$ 
```

---

explicit conditional expressions (e.g., convert `assume(x)` to `assume(x!=0)`). Second,  $\text{match}_{\hat{R}}(\pi, q, P)$  constructs a data-flow graph of  $P$ , and computes the slice of the graph that may reach  $q$ . Third, it builds an abstract data-flow graph from this slice. That is, it abstracts the atomic commands in the slice, merges nodes in the slice that are labeled with the same (abstract) atomic command, and induces arcs between nodes after merging in the standard way. Let  $(N_q, \rightsquigarrow_q)$  be the resulting abstract data-flow graph, and  $(N_0, \rightsquigarrow_0)$  the node and arc sets of  $\pi$ . In both cases, nodes are identified with their labels, so that  $N_q$  and  $N_0$  are the sets of (abstract) atomic commands. Finally,  $\text{match}_{\hat{R}}(\pi, q, P)$  returns 0 or 1 according to the criterion:  $\text{match}_{\hat{R}}(\pi, q, P) = 1 \iff N_0 \subseteq N_q \wedge (\rightsquigarrow_0) \subseteq (\rightsquigarrow_q^*)$ . The criterion means that the checking of our procedure succeeds if all the atomic commands in  $N_0$  appear in  $N_q$  and their dependencies encoded in  $\rightsquigarrow_0$  are respected by the transitive dependencies  $\rightsquigarrow_q^*$  in the query. Taking the transitive closure  $(\rightsquigarrow_q^*)$  here is important. It enables  $\text{match}_{\hat{R}}(\pi, q, P)$  to detect whether the programming pattern encoded in  $\pi$  appears somewhere in the program slice for  $q$ , even when the slice contains commands not related to the pattern.

## 5.4 Final Algorithm

Algorithm 1 shows the final algorithm for feature generation. It takes codebases  $\mathbf{P} = \{P_1, \dots, P_n\}$ , a static analysis  $F$  (Section 3), and a set  $R$  of grammar rules for the target programming language. Then, it returns the set  $\Pi$  of features.

The algorithm begins by splitting the codebases  $\mathbf{P}$  into a training set  $\mathbf{P}_{tr} = \{P_1, \dots, P_r\}$  and a validation set  $\mathbf{P}_{va} = \{P_{r+1}, \dots, P_n\}$  (line 1). In our experiments, we set  $r$  to the nearest integer to  $0.7n$ . Then, the algorithm calls `gen_fp` with  $\mathbf{P}_{tr}$  and the static analysis, so as to generate feature programs. Next, it initializes the score  $s_{best}$  to  $-1$ , and the set of features  $\Pi_{best}$  to the empty set. At lines 4–12, the algorithm repeatedly improves  $\Pi_{best}$  until it hits the limit of the given time budget. Recall that the performance

of our approach depends on a set  $\hat{R}$  of grammar rules, which determines how much atomic commands get abstracted. In each iteration of the loop, the algorithm chooses  $\hat{R} \subseteq R$  according to the strategy that we will explain shortly, and calls `build_dfg`( $\mathbf{P}_{feat}, \hat{R}$ ) to generate a new candidate set of features  $\Pi$ . Then, using this candidate set, the algorithm invokes an off-the-shelf learning algorithm (line 7) for learning an analysis heuristic  $\mathcal{H}_C$  from the training data  $\mathbf{P}_{tr}$ ; the subscript  $\mathcal{C}$  denotes a classifier built by the learning algorithm. The quality of the learned heuristic is evaluated on the validation set  $\mathbf{P}_{va}$  (line 8) by computing the  $F_1$ -score<sup>2</sup> of  $\mathcal{C}$ . If this evaluation gives a better score than the current best  $s_{best}$ , the set  $\Pi$  becomes a new current best  $\Pi_{best}$  (lines 9–10). To save computation, before running our algorithm, we run the static analysis  $F$  for all programs in the codebases  $\mathbf{P}$  with highest precision 1 and again with lowest precision 0, and record the results as labels for all queries in  $\mathbf{P}$ . This preprocessing lets us avoid calling  $F$  in `learn` and `evaluate`. As a result, after feature programs  $\mathbf{P}_{feat}$  are computed at line 2, building data-flow graphs and learning/evaluating the heuristic do not invoke the static analysis, so that each iteration of the loop in Algorithm 1 runs fast.

Our algorithm chooses a subset  $\hat{R} \subseteq R$  of grammar rules using a greedy bottom-up search. It partitions the grammar rules in Figure 2 into four groups  $R = R_1 \uplus R_2 \uplus R_3 \uplus R_4$  such that  $R_1$  contains the rules for the nonterminal  $c$  for commands,  $R_2$  those for the nonterminals  $e, lv$  for expressions,  $R_3$  the rules for the nonterminals  $\oplus, <$  for operators, and  $R_4$  those for the remaining nonterminals  $x, c$  for variables and constants. These sets form a hierarchy with  $R_i$  above  $R_{i+1}$  for  $i \in \{1, 2, 3\}$  in the following sense: for a typical derivation tree of the grammar, an instance of a rule in  $R_i$  usually appears nearer to the root of the tree than that of a rule in  $R_{i+1}$ . Algorithm 1 begins by choosing a subset of  $R_3$  randomly and setting the current rule set  $\hat{R}$  to the union of this subset and  $R_4$ . Including the rules in  $R_4$  has the effect of making the generated features (i.e., abstract data-flow graphs) forget variable names and constants that are specific to programs in the training set, so that they generalize well across different programs. This random choice is repeated for a fixed number of times (without choosing previously chosen abstractions), and the best  $\hat{R}_3$  in terms of its score  $s$  is recorded. Then, Algorithm 1 similarly tries different randomly-chosen subsets of  $R_2$  but this time using the best  $\hat{R}_3$  found, instead of  $R_4$ , as the set of default rules to include. The best choice  $\hat{R}_2$  is again recorded. Repeating this process with  $R_1$  and the best  $\hat{R}_2$  gives the final result  $\hat{R}_1$ , which leads to the result of Algorithm 1.

## 6. Instance Analyses

We have applied our feature-generation algorithm to three parametric program analyses: partially flow-sensitive inter-

<sup>2</sup>  $2 \cdot \text{precision} \cdot \text{recall} / (\text{precision} + \text{recall})$ .



val and pointer analyses, and a partial Octagon analysis. These analyses are designed following the data-driven approach of Section 4, so they are equipped with engines for learning analysis heuristics from given codebases. Our algorithm generates features required by these learning engines.

In this section, we describe the instance analyses. In these analyses, a program is given by its control-flow graph  $(\mathbb{C}, \hookrightarrow)$ , where each program point  $c \in \mathbb{C}$  is associated with an atomic command in Figure 2. We assume heap abstraction based on allocation sites and the existence of a variable for each site in a program. This lets us treat dynamically allocated memory cells simply as variables.

**Two Partially Flow-sensitive Analyses** We use partially flow-sensitive interval and pointer analyses that are designed according to the recipe in [40]. These analyses perform the sparse analysis [8, 13, 32, 39] in the sense that they work on *data-flow* graphs. Their flow-sensitivity is controlled by a chosen set of program variables; only the variables in the set are analyzed flow-sensitively. In terms of the terminologies of Section 3, the set of program components  $\mathbb{J}$  is that of variables  $\text{Var}$ , an analysis parameter  $\mathbf{a} \in \mathcal{A} = \{0, 1\}^{\mathbb{J}}$  specifies a subset of  $\text{Var}$ .

Both interval and pointer analyses define functions  $F: \mathbb{P} \times \mathcal{A} \rightarrow \wp(\mathbb{Q})$  that take a program and a set of variables and return proved queries in the program. They compute mappings  $D \in \mathbb{D} = \mathbb{C} \rightarrow \mathbb{S}$  from program points to abstract states, where an abstract state  $s \in \mathbb{S}$  itself is a map from program variables to values, i.e.,  $\mathbb{S} = \text{Var} \rightarrow \mathbb{V}$ . In the interval analysis,  $\mathbb{V}$  consists of intervals, and in the pointer analysis,  $\mathbb{V}$  consists of sets of the addresses of program variables.

Given an analysis parameter  $\mathbf{a}$ , the analyses compute the mappings  $D \in \mathbb{D}$  as follows. First, they construct a data-flow graph for variables in  $\mathbf{a}$ . For each program point  $c \in \mathbb{C}$ , let  $D(c) \subseteq \text{Var}$  and  $U(c) \subseteq \text{Var}$  be the definition and use sets. Using these sets, the analyses construct a data-flow relation  $(\rightsquigarrow_{\mathbf{a}}) \subseteq \mathbb{C} \times \text{Var} \times \mathbb{C}$ :  $c_0 \rightsquigarrow_{\mathbf{a}}^x c_n$  holds if there exists a path  $[c_0, c_1, \dots, c_n]$  in the control-flow graph such that  $x$  is defined at  $c_0$  (i.e.,  $x \in D(c_0)$ ) and used at  $c_n$  (i.e.,  $x \in U(c_n)$ ), but it is not re-defined at any of the intermediate points  $c_i$ , and the variable  $x$  is included in the parameter  $\mathbf{a}$ . Second, the analyses perform flow-insensitive analyses on the given program, and store the results in  $s_I \in \mathbb{S}$ . Finally, they compute fixed points of the function  $F_{\mathbf{a}}(D) = \lambda c. f_c(s')$  where  $f_c$  is a transfer function at a program point  $c$ , and the abstract state  $s'$  is the following combination of  $D$  and  $s_I$  at  $c$ :  $s'(x) = s_I(x)$ , for  $x \notin \mathbf{a}$  and for  $x \in \mathbf{a}$ ,  $s'(x) = \bigsqcup_{c_0 \rightsquigarrow_{\mathbf{a}}^x c} D(c_0)(x)$ . Note that for variables not in  $\mathbf{a}$ ,  $F_{\mathbf{a}}$  treats them flow-insensitively by using  $s_I$ . When  $\mathbf{a} = \text{Var}$ , the analyses become ordinary flow-sensitive analyses, and when  $\mathbf{a} = \emptyset$ , they are just flow-insensitive analyses.

**Partial Octagon Analysis** We use the partial Octagon analysis formulated in [22]. Let  $m$  be the number of variables in the program, and write  $\text{Var} = \{x_1, \dots, x_m\}$ . The set of program components  $\mathbb{J}$  is  $\text{Var} \times \text{Var}$ , so an analysis param-

eter  $\mathbf{a} \in \mathcal{A} = \{0, 1\}^{\mathbb{J}}$  consists of pairs of program variables. Intuitively,  $\mathbf{a}$  specifies which two variables should be tracked together by the analysis. Given such  $\mathbf{a}$ , the analysis defines the smallest partition  $\Gamma$  of variables such that every  $(x, y) \in \mathbf{a}$  is in the same partition of  $\Gamma$ . Then, it defines a grouped Octagon domain  $\mathbb{O}_{\Gamma} = \prod_{\gamma \in \Gamma} \mathbb{O}_{\gamma}$  where  $\mathbb{O}_{\gamma}$  is the usual Octagon domain for the variables in the partition  $\gamma$ . The abstract domain of the analysis is  $\mathbb{C} \rightarrow \mathbb{O}_{\Gamma}$ , the collection of maps from program points to grouped Octagons. The analysis performs fixed-point computation on this domain using adjusted transfer functions of the standard Octagon analysis. The details can be found in [22].

We have to adjust the learning engine and our feature-generation algorithm for this partial Octagon slightly. This is because the full Octagon analysis on a program  $P$  (that is,  $F(P, \mathbf{1})$ ) does not work usually when the size of  $P$  is large ( $\geq 20\text{KLOC}$  in our experiments). Whenever the learning part in Section 4 and our feature-generation algorithm have to run  $F(P, \mathbf{1})$  originally, we run the impact pre-analysis in [22] instead. This pre-analysis is a fully relational analysis that works on a simpler abstract domain than the full Octagon, and estimates the behavior of the full Octagon; it defines a function  $F^{\sharp}: \mathbb{P} \rightarrow \wp(\mathbb{Q})$ , which takes a program and returns a set of queries in the program that are likely to be verified by the full Octagon. Formally, we replaced the predicate  $\phi$  in Section 5.1 by  $\phi_q(P) = (q \notin F(P, \mathbf{0}) \wedge q \in F^{\sharp}(P))$ .

## 7. Experiments

We evaluated our feature-generation algorithm with the instance analyses in Section 6. We used the interval and Octagon analyses for proving the absence of buffer overruns, and the pointer analysis the absence of null dereferences.

The three analyses are implemented on top of our analysis framework for the C programming language [38]. The framework provides a baseline analysis that uses heap abstraction based on allocation sites and array smashing, is field-sensitive but context-insensitive, and performs the sparse analysis [8, 13, 32, 39]. We extended this baseline analysis to implement the three analyses. Our pointer analysis uses Andersen’s algorithm [3]. The Octagon analysis is implemented by using the OptOctagons and Apron libraries [24, 50]. Our implementation of the feature-generation algorithm in Section 5 and the learning part in Section 4 is shared by the three analyses except that the analyses use slight variants of the req function in Section 4, which converts a query to program components. In all three analyses, req first computes a dependency slice of a program for a given query. Then, it collects program variables in the slice for the interval and pointer analyses, and pairs of all program variables in the slice for the Octagon analysis. The computation of the dependency slice is approximate in that it estimates dependency using a flow-insensitive pointer analysis and ignores atomic commands too far away from the

query when the size of the slice goes beyond a threshold.<sup>3</sup> This approximation ensures that the cost of computing req is significantly lower than that of the main analyses. We use the same dependency analysis in match. Our implementation uses C-Reduce [46] for generating feature programs and a decision tree algorithm [42] for learning a classifier for queries. Our evaluation aims at answering four questions:

- **Effectiveness:** Does our feature-generation algorithm enable the learning of good analysis heuristics?
- **Comparison with manually-crafted features:** How does our approach of learning with automatically-generated features compare with the existing approaches of learning with manually-crafted features?
- **Impact of reducing and learning:** Does reducing a program really help for generating good features? Given a set of features, does learning lead to a classifier better than a simple (disjunctive) pattern matcher?
- **Generated features:** Does our feature-generation algorithm produce informative features?

**Effectiveness** We compared the performance of our three instance analyses with their standard counterparts:

- Flow-insensitive(FI1) & -sensitive(FS1) interval analyses
- Flow-insensitive(FIP) & -sensitive(FSP) pointer analyses
- Flow-sensitive interval analysis (FS1) and partial Octagon analysis by impact pre-analysis (IMPCT) [37].

We did not include the Octagon analysis [33] in the list because the analysis did not scale to medium-to-large programs in our benchmark set.

In experiments, we used 60 programs (ranging 0.4–109.6 KLOC) collected from Linux and GNU packages. The programs are shown in Table 4. To evaluate the performance of learned heuristics for the interval and pointer analyses, we randomly partitioned the 60 programs into 42 training programs (for feature generation and learning) and 18 test programs (for cross validation). For the Octagon analysis, we used only 25 programs out of 60 because for some programs, Octagon and the interval analysis prove the same set of queries. We selected these 25 by running the impact pre-analysis [22] for the Octagon on all the 60 programs and choosing the ones that may benefit from Octagon according to the results of this pre-analysis. The 25 programs are shown in Table 5. We randomly partitioned the 25 programs into 17 training programs and 8 test programs. From the training programs, we generated features and learned a heuristic based on these features.<sup>4</sup> The learned heuristic was used for analyzing the test programs. We repeated this procedure for five times with different partitions of the whole

<sup>3</sup> In our implementation, going beyond a threshold means having more than 200 program variables.

<sup>4</sup> We followed the practice used in representation learning [6], where both feature generation and learning are done with the same dataset.

program sets. The average numbers of generated features over the five trials were 38 (interval), 45 (pointer), and 44 (Octagon). C-Reduce took 0.5–24 minutes to generate a feature program from a query. All experiments were done on a Ubuntu machine with Intel Xeon cpu (2.4GHz) and 192GB of memory.

Table 1 shows the performance of the learned heuristics on the test programs for the interval analysis. The learned classifier for queries (Section 4) was able to select 75.7% of FS-provable but FI-unprovable queries on average (i.e., 75.7% recall) and 74.5% of the selected queries were actually proved under FS only (i.e., 74.5% precision). With the analysis heuristic on top of this classifier, our partially flow-sensitive analysis could prove 80.2% of queries that require flow-sensitivity while increasing the time of the flow-insensitive analysis by 2.0x on average. We got 80.2 (higher than 75.7) because the analysis parameter is the set of all the program components for queries selected by the classifier and this parameter may make the analysis prove queries not selected by the classifier. The fully flow-sensitive analysis increased the analysis time by 46.5x. We got similar results for the other two analyses (Tables 2 and 3).

**Comparison with Manually-Crafted Features** We compared our approach with those in Oh et al. [40] and Heo et al. [22], which learn analysis heuristics using manually-crafted features. The last two columns of Table 1 present the performance of the partially flow-sensitive interval analysis in [40], and those of Table 3 the performance of the partial Octagon analysis in [22].

The five trials in the tables use the splits of training and test programs in the corresponding entries of Tables 1 and 3. Our implementation of Oh et al.’s approach used their 45 manually-crafted features, and applied their Bayesian optimization algorithm to our benchmark programs. Their approach requires the choice of a threshold value  $k$ , which determines how many variables should be treated flow-sensitively. For each trial and each program in that trial, we set  $k$  to the number of variables selected by our approach, so that both approaches induce similar overhead in analysis time. Our implementation of Heo et al.’s approach used their 30 manually-crafted features, and applied their supervised learning algorithm to our benchmark programs.

The results show that our approach is on a par with the existing ones, while not requiring the manual feature design. For the interval analysis, our approach consistently proved more queries than Oh et al.’s (80.2% vs 55.1% on average). For Octagon, Heo et al.’s approach proved more queries than ours (81.1% vs 96.2%). We warn a reader that these are just end-to-end comparisons and it is difficult to draw a clear conclusion, as the learning algorithms of the three approaches are different. However, the overall results show that using automatically-generated features is as competitive as using features crafted manually by analysis designers.

Trial	Query Prediction		#Proved Queries			Analysis Cost (sec)			Quality		Oh et al. [40]	
	Precision	Recall	FII (a)	FSI (b)	Ours (c)	FII (d)	FSI	Ours (e)	Prove	Cost	Prove	Cost
1	71.5 %	78.8 %	6,537	7,126	7,019	26.7	569.0	52.0	81.8 %	1.9x	56.6 %	2.0x
2	60.9 %	75.1 %	4,127	4,544	4,487	58.3	654.2	79.9	86.3 %	1.4x	49.2 %	2.4x
3	78.2 %	74.0 %	6,701	7,532	7,337	50.9	6,175.2	167.5	76.5 %	3.3x	51.1 %	3.4x
4	72.9 %	76.1 %	4,399	4,956	4,859	36.9	385.1	44.9	82.6 %	1.2x	54.8 %	1.2x
5	83.2 %	75.3 %	5,676	6,277	6,140	31.7	1,740.3	61.6	77.2 %	1.9x	65.6 %	1.8x
TOTAL	74.5 %	75.7 %	27,440	30,435	29,842	204.9	9,523.9	406.1	<b>80.2 %</b>	<b>2.0x</b>	55.1 %	2.3x

**Table 1.** Effectiveness of partially flow-sensitive interval analysis. Quality: Prove =  $(c - a)/(b - a)$ , Cost =  $e/d$

Trial	Query Prediction		#Proved Queries			Analysis Cost (sec)			Quality	
	Precision	Recall	FIP	FSP	Ours	FIP	FSP	Ours	Prove	Cost
1	79.1 %	76.8 %	4,399	6,346	6,032	48.3	3,705.0	150.0	83.9 %	3.1x
2	78.3 %	77.1 %	7,029	8,650	8,436	48.9	651.4	74.0	86.8 %	1.5x
3	74.5 %	75.0 %	8,781	10,352	10,000	41.5	707.0	59.4	77.6 %	1.4x
4	73.8 %	75.9 %	10,559	12,914	12,326	51.1	4,107.0	164.3	75.0 %	3.2x
5	78.0 %	82.5 %	4,205	5,705	5,482	23.0	847.2	56.7	85.1 %	2.5x
TOTAL	76.6 %	77.3 %	34,973	43,967	42,276	212.9	10,017.8	504.6	<b>81.2 %</b>	<b>2.4x</b>

**Table 2.** Effectiveness of partially flow-sensitive pointer analysis

Trial	Query Prediction		#Proved Queries			Analysis Cost (sec)			Quality		Heo et al. [22]	
	Precision	Recall	FSI	IMPCT	Ours	FSI	IMPCT	Ours	Prove	Cost	Prove	Cost
1	74.8 %	81.3 %	3,678	3,806	3,789	140.7	389.8	230.5	86.7 %	1.6 x	100.0 %	3.0 x
2	84.1 %	82.6 %	5,845	6,004	5,977	613.5	18,022.9	782.9	83.0 %	1.3 x	94.3 %	1.8 x
3	82.8 %	73.0 %	1,926	2,079	2,036	315.2	2,396.9	416.0	71.9 %	1.3 x	92.2 %	1.1 x
4	77.6 %	85.2 %	2,221	2,335	2,313	72.7	495.1	119.9	80.7 %	1.6 x	100.0 %	2.0 x
5	71.6 %	78.4 %	2,886	2,962	2,944	148.9	557.2	209.7	76.3 %	1.4 x	96.1 %	2.3 x
TOTAL	79.0 %	79.9 %	16,556	17,186	17,067	1,291.0	21,861.9	1,759.0	<b>81.1 %</b>	<b>1.4 x</b>	96.2 %	1.8 x

**Table 3.** Effectiveness of partial Octagon analysis

**Impact of Reducing and Learning** In order to see the role of a reducer in our approach, we generated feature programs without calling the reducer in our experiment with the interval analysis. These unreduced feature programs were then converted to abstract data-flow graphs or features, which enabled the learning of a classifier for queries. The generated features were too specific to training programs, and the learned classifier did not generalize well to unseen test programs; removing a reducer dropped the average recall of the classifier from 75.7% to 58.2% for test programs.

In our approach, a feature is a reduced and abstracted program slice that illustrates when high precision of an analysis is useful for proving a query. Thus, one natural approach is to use the disjunction of all features as a classifier for queries. Intuitively, this classifier attempts to pattern-match each of these features against a given program and a query, and it returns true if some attempt succeeds. We ran our experiment on the interval analysis with this disjunctive classifier, instead of the original decision tree learned from training programs. This change of the classifier increased the recall from 75.7% to 79.6%, but dropped the precision significantly from 74.5% to 10.4%. The result shows the benefit of going beyond the simple disjunction of features and using a more sophisticated boolean combination of them (as encoded by a decision tree). One possible explanation is that

the matching of multiple features suggests the high complexity of a given program, which typically makes the analysis lose much information even under the high-precision setting.

**Generated Features** We ranked generated features in our experiments according to their Gini scores [9] in the learned decision tree, which measure the importance of these features for prediction. For each instance analysis, we show two features that rank consistently high in the five trials of experiments. For readability, we present them in terms of their feature programs, rather than as abstract data-flow graphs.

The top-two feature programs for the interval analysis and the pointer analysis are:

```
// Feature Program 1 for Interval
int buf[10];
for (i=0;i<7;i++) { buf[i]=0; /* Query */ }

// Feature Program 2 for Interval
i=255; p=malloc(i);
while (i>0) { *(p+i)=0; /* Query */ i--; }

// Feature Program 1 for Pointer
i=128; p=malloc(i);
if (p==0) return; else *p=0; /* Query */

// Feature Program 2 for Pointer
p=malloc(i); p=&a; *p=0; /* Query */
```

The feature programs for the interval analysis describe cases where a consecutive memory region is accessed in a loop through increasing or decreasing indices and these indices are bounded by a constant from above or from below because of a loop condition. The programs for the pointer analysis capture cases where the safety of pointer dereference is guaranteed by null check or preceding strong update. All of these programs are typical showcases of flow-sensitivity.

The top-two feature programs for the partial Octagon analysis are:

```
// Feature Program 1 for Octagon
size=POS_NUM; arr=malloc(size);
arr[size-1]=0; /* Query */

// Feature Program 2 for Octagon
size=POS_NUM; arr=malloc(size);
for(i=0;i<size;i++){ arr[i]=0; /* Query */ }
```

These feature programs allocate an array of a positive size and access the array using an index that is related to this size in a simple linear way. They correspond to our expectation about when the Octagon analysis is more effective than the interval analysis.

When converting feature programs to abstract data-flow graphs, our approach automatically identifies the right abstraction level of commands for each instance analysis (Algorithm 1). In the interval analysis, the abstraction found by our approach was to merge all the comparison operators (e.g.,  $<$ ,  $\leq$ ,  $>$ ,  $\geq$ ,  $=$ ) but to differentiate all the arithmetic operators (e.g.,  $+$ ,  $-$ ,  $*$ ). This is because, in the interval analysis, comparison with constants is generally a good signal for improving precision regardless of a specific comparison operator used, but the analysis behaves differently when analyzing different commands involving  $+$  or  $-$ . With other abstractions, we obtained inferior performance; for example, when we differentiate comparison operators and abstract away arithmetic operators, the recall was dropped from 75.7% to 54.5%. In the pointer analysis, the best abstraction was to merge all arithmetic and comparison operators while differentiating equality operators ( $=$ ,  $\neq$ ). In the Octagon analysis, our approach identified an abstraction that merges all comparison and binary operators while differentiating addition/subtraction operators from them.

**Limitations** Our current implementation has two limitations. First, because we approximately generate data dependency (in `req` and `match`) within a threshold, we cannot apply our approach to instances that require computing long dependency chains, e.g., context-sensitive analysis. We found that computing dependency slices of queries beyond procedure boundaries efficiently with enough precision is hard to achieve in practice. Second, our method could not be applicable to program analyses for other programming languages (e.g., oop, functional), as we assume that a powerful program reducer and a way to efficiently build data-flow

graphs exist for the target language, which does not hold for, e.g., JavaScript.

## 8. Related Work

**Parametric Static Analysis** In the past decade, a large amount of research in static analysis has been devoted for developing an effective heuristic for finding a good abstraction. Several effective techniques based on counterexample-guided abstraction refinement (CEGAR) [4, 11, 17, 19, 21, 53, 54] have been developed, which iteratively refine the abstraction based on the feedback from the previous runs. Other techniques choose an abstraction using dynamic analyses [20, 35] or pre-analyses [37, 51]. Or they simply use a good manually-designed heuristic, such as the one for controlling the object sensitivity for Java programs [25]. In all of these techniques, heuristics for choosing an abstraction cannot automatically extract information from one group of programs, and generalize and apply it to another group of programs. This cross-program generalization in those techniques is, in a sense, done manually by analysis designers.

Recently, researchers have proposed new techniques for finding effective heuristics automatically rather than manually [10, 18, 22, 40]. In these techniques, heuristics themselves are parameterized by hyperparameters, and an effective hyperparameter is learned from existing codebases by machine learning algorithms, such as Bayesian optimization and decision tree learning [10, 22, 40]. In [18], a learned hyperparameter determines a probabilistic model, which is then used to guide an abstraction-refinement algorithm.

Our work improves these recent techniques by addressing the important issue of feature design. The current learning-based techniques assume well-designed features, and leave the obligation of discharging this nontrivial assumption to analysis designers [10, 22, 40]. The only exception is [18], but the technique there applies only to a specific class of program analyses written in Datalog. In particular, it does not apply to the analyses with infinite abstract domains, such as interval and Octagon analyses. Our work provides a new automatic way of discharging the assumption on feature design, which can be applicable to a wide class of program analyses because our approach uses program analysis as a black box and generates features (i.e., abstracted small programs) not tied to the internals of the analysis.

### *Application of Machine Learning in Program Analysis*

Recently machine learning techniques have been applied for addressing challenging problems in program analysis. They have been used for generating candidate invariants from data collected from testing [14, 15, 36, 48, 49], for discovering intended behaviors of programs (e.g., preconditions of functions, API usages, types, and informative variable names) [1, 5, 16, 26, 27, 30, 34, 41, 44, 47, 55, 56], for finding semantically similar pieces of code [12], and for synthesizing programs (e.g., code completion and patch generation) [2, 7, 23, 31, 43, 45]. Note that the problems solved by

these applications are different from ours, which is to learn good analysis heuristics from existing codebases and to generate good features that enable such learning.

**Feature Learning in Machine Learning** Our work can be seen as a feature learning technique specialized to the program-analysis application. Automating the feature-design process has been one of the holy grails in the machine learning community, and a large body of research has been done under the name of representation learning or feature learning [6]. Deep learning [28] is perhaps the most successful feature-learning method, which simultaneously learns features and classifiers through multiple layers of representations. It has been recently applied to programming tasks (e.g. [2]). A natural question is, thus, whether deep learning can be used to learn program analysis heuristics as well. In fact, we trained a character-level convolutional network in Zhang et al. [52] for predicting the need for flow-sensitivity in interval analysis. We represented each query by the 300 characters around the query in the program text, and used pairs of character-represented query and its provability as training data. We tried a variety of settings (varying, e.g., #layers, width, #kernels, kernel size, activation functions, #output units, etc) of the network, but the best performance we could achieve was 93% of recall with disappointing 27% of precision on test data. Achieving these numbers was highly nontrivial, and we could not find intuitive explanation about why a particular setting of the network leads to better results than others. We think that going beyond 93% recall and 27% precision in this application is challenging and requires expertise in deep learning.

## 9. Conclusion

We have presented an algorithm that mines existing codebases and generates features for a data-driven parametric static analysis. The generated features enable the learning of an analysis heuristic from the codebases, which decides whether each part of a given program should be analyzed under high precision or under low precision. The key ideas behind the algorithm are to use abstracted code snippets as features, and to generate such snippets using a program reducer. We applied the algorithm to partially flow-sensitive interval and pointer analyses and partial Octagon analysis. Our experiments with these analyses and 60 programs from Linux and GNU packages show that the learned heuristics with automatically-generated features achieve performance comparable to those with manually-crafted features.

Designing a good set of features is a nontrivial and costly step in most applications of machine learning techniques. We hope that our algorithm for automating this feature design for data-driven program analyses or its key ideas help attack this feature-design problem in the ever-growing applications of machine learning for program analysis, verification, and other programming tasks.

## References

- [1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *FSE*, 2015.
- [2] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. A convolutional attention network for extreme summarization of source code. In *ICML*, 2016.
- [3] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).
- [4] T. Ball and S. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL*, 2002.
- [5] Nels E. Beckman and Aditya V. Nori. Probabilistic, modular and scalable inference of tpestate specifications. In *PLDI*, pages 211–221, 2011.
- [6] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis Machine Intelligence*, 35(8), August 2013.
- [7] Pavol Bielik, Veselin Raychev, and Martin Vechev. PHOG: probabilistic model for code. In *ICML*, 2016.
- [8] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Selective control-flow abstraction via jumping. In *OOPSLA*, 2015.
- [9] Leo Breiman. Random Forests. *Machine Learning*, 2001.
- [10] Sooyoung Cha, Seun Jeong, and Hakjoo Oh. Learning a strategy for choosing widening thresholds from a large codebase. In *APLAS*, 2016.
- [11] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *JACM*, 50(5), 2003.
- [12] Yaniv David, Nimrod Partush, and Eran Yahav. Statistical similarity of binaries. In *PLDI*, 2016.
- [13] Azadeh Farzan and Zachary Kincaid. Verification of parameterized concurrent programs by modular reasoning about data and control. In *POPL*, 2012.
- [14] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. Ice: A robust framework for learning invariants. In *CAV*, 2014.
- [15] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. Learning invariants using decision trees and implication counterexamples. In *POPL*, 2016.
- [16] Timon Gehr, Dimitar Dimitrov, and Martin Vechev. Learning commutativity specifications. In *CAV*, 2015.
- [17] S. Grebenshchikov, A. Gupta, N. Lopes, C. Popeea, and A. Rybalchenko. HSF(C): A software verifier based on Horn clauses. In *TACAS*, 2012.
- [18] Radu Grigore and Hongseok Yang. Abstraction refinement guided by a learnt probabilistic model. In *POPL*, 2016.
- [19] B. Gulavani, S. Chakraborty, A. Nori, and S. Rajamani. Automatically refining abstract interpretations. In *TACAS*, 2008.
- [20] Ashutosh Gupta, Rupak Majumdar, and Andrey Rybalchenko. From tests to proofs. *STTT*, 15(4):291–303, 2013.

- [21] T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *POPL*, 2004.
- [22] Kihong Heo, Hakjoo Oh, and Hongseok Yang. Learning a variable-clustering strategy for Octagon from labeled data generated by a static analysis. In *SAS*, 2016.
- [23] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *ICSE*, 2012.
- [24] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, pages 661–667, 2009.
- [25] George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *PLDI*, 2013.
- [26] Omer Katz, Ran El-Yaniv, and Eran Yahav. Estimating types in binaries using predictive modeling. In *POPL*, 2016.
- [27] Sulekha Kulkarni, Ravi Mangal, Xin Zhang, and Mayur Naik. Accelerating program analyses by cross-program training. In *OOPSLA*, 2016.
- [28] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.
- [29] Percy Liang, Omer Tripp, and Mayur Naik. Learning minimal abstractions. In *POPL*, 2011.
- [30] V. Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. Merlin: specification inference for explicit information flow problems. In *PLDI*, 2009.
- [31] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *POPL*, 2016.
- [32] Magnus Madsen and Anders Møller. Sparse dataflow analysis with pointers and reachability. In *SAS*, 2014.
- [33] A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
- [34] Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based semantic code search over partial programs. In *OOPSLA*, pages 997–1016, 2012.
- [35] Mayur Naik, Hongseok Yang, Ghila Castelnovo, and Mooly Sagiv. Abstractions from tests. In *POPL*, 2012.
- [36] Aditya V. Nori and Rahul Sharma. Termination proofs from tests. In *FSE*, 2013.
- [37] Hakjoo Oh, , Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective context-sensitivity guided by impact pre-analysis. In *PLDI*, 2014.
- [38] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Sparrow. <http://ropas.snu.ac.kr/sparrow>.
- [39] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for C-like languages. In *PLDI*, 2012.
- [40] Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. Learning a strategy for adapting a program analysis via Bayesian optimisation. In *OOPSLA*, 2015.
- [41] Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. In *PLDI*, 2016.
- [42] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine Learning in Python. *The Journal of Machine Learning Research*, 2011.
- [43] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. In *POPL*, 2016.
- [44] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from “big code”. In *POPL*, 2015.
- [45] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *PLDI*, 2014.
- [46] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *PLDI*, 2012.
- [47] Sriram Sankaranarayanan, Swarat Chaudhuri, Franjo Ivančić, and Aarti Gupta. Dynamic inference of likely data preconditions over predicates by tree learning. In *ISSTA*, 2008.
- [48] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. A data driven approach for algebraic loop invariants. In *ESOP*, 2013.
- [49] Rahul Sharma, Aditya V. Nori, and Alex Aiken. Interpolants as classifiers. In *CAV*, 2012.
- [50] Gagandeep Singh, Markus Püschel, and Martin Vechev. Making numerical program analysis fast. In *PLDI*, 2015.
- [51] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *PLDI*, 2014.
- [52] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *NIPS*, 2015.
- [53] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. On abstraction refinement for program analyses in datalog. In *PLDI*, 2014.
- [54] Xin Zhang, Mayur Naik, and Hongseok Yang. Finding optimum abstractions in parametric dataflow analysis. In *PLDI*, 2013.
- [55] He Zhu, Aditya V. Nori, and Suresh Jagannathan. Learning refinement types. In *ICFP*, 2015.
- [56] He Zhu, Gustavo Petri, and Suresh Jagannathan. Automatically learning shape specifications. In *PLDI*, 2016.

## A. Benchmark Programs

Table 4 and 5 show the benchmark programs for the partially flow-sensitive interval and pointer analyses, and the partial Octagon analysis, respectively.

Programs	LOC	Programs	LOC
brutefir-1.0f	398	mpage-2.5.6	14,827
consol_calculator	1,124	bc-1.06	16,528
dtmf-dial-0.2+1	1,440	ample-0.5.7	17,098
id3-0.15	1,652	irmp3-ncurses-0.5.3.1	17,195
polymorph-0.4.0	1,764	tnef-1.4.6	18,172
unhtml-2.3.9	2,057	ecasound2.2-2.7.0	18,236
spell-1.0	2,284	gzip-1.2.4a	18,364
mp3rename-0.6	2,466	unrtf-0.19.3	19,019
mp3wrap-0.5	2,752	jwhois-3.0.1	19,375
ncompress-4.2.4	2,840	archimedes	19,552
pgdbf-0.5.0	3,135	aewan-1.0.01	28,667
mcf-spec2000	3,407	tar-1.13	30,154
acpi-1.4	3,814	normalize-audio-0.7.7	30,984
unsort-1.1.2	4,290	less-382	31,623
checkmp3-1.98	4,450	tmndec-3.2.0	31,890
cam-1.05	5,459	gbsplay-0.0.91	34,002
bottlerocket-0.05b3	5,509	flake-0.11	35,951
129.compress	6,078	enscript-1.6.5	38,787
e2ps-4.34	6,222	twolame-0.3.12	48,223
httptunnel-3.3	7,472	mp3c-0.29	52,620
mpegdemux-0.1.3	7,783	bison-2.4	59,955
barcode-0.96	7,901	tree-puzzle-5.2	62,302
stripcc-0.2.0	8,914	icecast-server-1.3.12	68,571
xfpt-0.07	9,089	dico-2.0	69,308
man-1.5h1	11,059	aalib-1.4p5	73,413
cjet-0.8.9	11,287	pies-1.2	84,649
admesh-0.95	11,439	rnv-1.7.10	93,858
hspell-1.0	11,520	mpg123-1.12.1	101,701
juke-0.7	12,518	raptor-1.4.21	109,053
gzip-spec2000	12,980	lsh-2.0.4	109,617

**Table 4.** 60 benchmark programs for our partially flow-sensitive interval and pointer analyses.

Programs	LOC	Programs	LOC
brutefir-1.0f	398	ecasound2.2-2.7.0	18,236
consol_calculator	1,124	unrtf-0.19.3	19,019
dtmf-dial-0.2+1	1,440	jwhois-3.0.1	19,375
id3-0.15	1,652	less-382	31,623
spell-1.0	2,284	flake-0.11	35,951
mp3rename-0.6	2,466	mp3c-0.29	52,620
e2ps-4.34	6,222	bison-2.4	59,955
httptunnel-3.3	7,472	icecast-server-1.3.12	68,571
mpegdemux-0.1.3	7,783	dico-2.0	69,308
barcode-0.96	7,901	pies-1.2	84,649
juke-0.7	12,518	raptor-1.4.21	109,053
bc-1.06	16,528	lsh-2.0.4	109,617
irmp3-ncurses-0.5.3.1	17,195		

**Table 5.** 25 benchmark programs for our partial Octagon analysis.